Emacs Predictive Completion Manual

Version 0.23

Toby Cubitt

This manual documents the Emacs Predictive Completion package, version 0.23

Table of Contents

1 What is predictive completion?

The languages we use to communicate contain a large amount of redundancy. Given the first few letters of a word, for instance, it's not too difficult to predict what should come next. Try it! You can probably easily guess how to fill in the missing letters in the following sentence:

> Giv th fir fe lett o a wor i no diffi t predi wh shou com nex.

This is even more true of the languages used to communicate with computers, which typically have very restricted vocabularies and rigidly defined grammars. Redundancy occurs on many levels: on the level of individual characters (as illustrated above), on the level of words (we can often predict quite accurately what words are likely to come next in a sentence, based on grammar and usage), and perhaps even on higher levels. Predictive completion exploits this redundancy by attempting to complete what you are trying to type before you've finished typing it.

The predictive completion package described here is an add-on to the GNU Emacs editor, which implements a new minor-mode called `predictive-mode` (see Section "Minor Modes" in *GNU Emacs Manual*). When this predictive completion minor-mode is switched on, Emacs will try to complete the word you are typing based on the characters typed so far. As you add characters, it can look up words starting with those characters in a dictionary, and offer to insert the most likely ones. How you choose which completion (if any) to insert depends on various customization settings, some more intrusive than others.

Note that by only taking into account characters belonging to the current word when predicting how to complete it, much of the redundancy in language remains unexploited. This limitation is partly for simplicity, but also because some of the benefits of predictive completion would be lost if context (i.e. the preceding words and sentences) was also taken into account.

What benefits does predictive completion bring?

1. Saving on typing (obviously!): you don't have to type the whole word.
2. Automatic spelling assistance: when you type the first few characters of a word, only correct spellings of the whole word are offered. This is not only useful for human languages. The predictive completion mode can be set up to complete on variable and function names in a program, helping avoid bugs due to misspelled names.
3. Faster typing (possibly): not quite the same as point 1. As you get used to predictive completion, your fingers will start to learn the key sequences required for frequently used words. And these key sequences will usually be much shorter than typing the full word. (This is one reason for not taking context into account. If we did, the same word might require different key sequences depending on the words and sentences preceding it.)

Whether predictive mode really speeds up typing or not is debatable. At the very least it depends on the context in which predictive mode is used. Although you have to type less, you have to process more information as you type, to decide whether to accept the offered completion or whether to add more characters to narrow down the completions to word you want. And this increased "cognitive load" might wipe out the advantage of having to type fewer characters. Studies indicate that the cognitive does increase, but whether or not this

negates all speed advantage to predictive completion systems is less clear. It seems plausible that the more you use predictive mode, the better you will become at quickly evaluating the possible completions. Also, with the default settings, there is nothing to stop you typing normally and ignoring the completions entirely until you want to make use of them. In the end, the only way to find out is to try it and see!

2 Obtaining and Installing

The current version of the predictive completion package can be obtained from `http://www.dr-qubit.org/emacs.php`.

If you are upgrading from a previous version, make sure you first backup dictionaries you use by dumping the words and data they contain to files (unless you want to start from scratch). To make the backups, use `predictive-dump-dict-to-file`. See Section 5.2 [Loading and Saving Dictionaries], page 22. It is *not* enough to make a copy of the dictionary file. Only the plain-text "dumped" format is guaranteed to be portable across different versions of predictive mode.

However, this version of the predictive completion package (version 0.22 and newer, or more specifically version 0.12 and newer of the '`dict-tree.el`' library) uses a different format than earlier versions, *even* for the "dumped" files! To port a dictionary from older versions to this version, use the following procedure:

1. backup the dictionay using `predictive-dump-dict-to-file` as just described, *using the old version of the predictive package*;
2. start Emacs;
3. load the '`predictive-convert-dump-format`' library (included in this version of the predictive completion package) using *`M-x load-file /path/to/predictive-convert-dump-format.el`*;
4. open the "dumped" dictionary backup file in Emacs;
5. run the `predictive-convert-dump-format` command;
6. recreate the dictionary using the `predictive-create-dict` command (see Section 5.1 [Creating Dictionaries], page 20).

Repeat this procedure for each dictionary that you want to port to this new version.

Older versions of the predictive completion package saved dictionaries in a compiled form that was not portable across different versions of Emacs. Newer versions now automatically save dictionaries in both compiled and uncompiled forms. If you are upgrading to a different version of Emacs, but continuing to use the same recent version of the predictive completion package, then it is sufficient to delete the compiled dictionary files (extension '`.elc`', but make sure you do *not* delete the uncompiled files with extension '`.el`'!), and let predictive mode recompile them automatically when the dictionaries are saved.

However, backing up your dictionaries by dumping them to plain-text files, as described above, is still a very good idea (just to be on the safe side!).

To install the package, extract the files using *`tar -xvzf predictive.tar.gz`*. This will unpack the files in a new directory called '`predictive/`'. Now byte-compile all the Lisp files and create the required dictionaries by running *`make`* in the '`predictive/`' directory. If necessary, you can specify explicitly where your Emacs resides with *`make EMACS=/path/to/emacs`*.

Then run *`make install`* to copy the package files to '`~/.emacs.d/predictive/`' (the '`~/.emacs.d/`' directory will be created if it doesn't already exist). If you want to install the files elsewhere, you can change the default install location using *`make install DESTDIR=/path/to/install/`* and the package files will be copied to the '`/path/to/install/`' directory.

The dictionaries are intended to be user-specific, and *must* be writable by your user. If you're performing a site-wide installation of the predictive package, the dictionary files (all files starting with the prefix '`dict-`' and ending in '`.elc`' or '`.el`', *except* '`dict-tree.el`') need to be copied to a separate writable directory for each user. For this reason, the dictionaries will still be installed under '`~/.emacs.d/predictive/`' even if *`DESTDIR`* is specified. (You will have to copy them manually for any other users who want to use predictive.) If you want to change the dictionary install location, use *`make install DICTDIR=/path/to/dictionaries/`*. This can of course be combined with the *`DESTDIR`* option: *`make install DESTDIR=/path/to/install/ DICTDIR=/path/to/dictionaries/`*.

Finally, put the following lines in your .emacs file (replace the paths as appropriate if you're not using the default install locations):

```
;; predictive install location
(add-to-list 'load-path "~/.emacs.d/predictive/")
;; dictionary locations
(add-to-list 'load-path "~/.emacs.d/predictive/latex/")
(add-to-list 'load-path "~/.emacs.d/predictive/texinfo/")
(add-to-list 'load-path "~/.emacs.d/predictive/html/")
;; load predictive package
(require 'predictive)
```

Alternatively, you can save memory and only load the lisp libraries when they're needed (i.e. when you first run the `predictive-mode` command), by replacing the final line as follows:

```
;; predictive install location
(add-to-list 'load-path "~/.emacs.d/predictive/")
;; dictionary locations
(add-to-list 'load-path "~/.emacs.d/predictive/latex/")
(add-to-list 'load-path "~/.emacs.d/predictive/texinfo/")
(add-to-list 'load-path "~/.emacs.d/predictive/html/")
;; load predictive package
(autoload 'predictive-mode "~/.emacs.d/predictive/predictive"
          "Turn on Predictive Completion Mode." t)
```

If you want to install the info documentation into your local info system, you should run the following command *as root*: *`make info-install`*. If your info '`dir`' file is not located in '`/usr/share/info/`', then you can specify its location using *`make info-install INFODIR=/path/to/info/`*.

For more details, and for alternative installation options, you should consult the '`INSTALL`' file included in the predictive package.

3 Quick-Start

This "Quick-Start" describes the default behaviour. However, predictive completion mode can be heavily customized, allowing its behaviour to be radically changed. The many and various predictive completion mode customization options can be found in the `predictive` and `completion-ui` customization groups. If you like the idea of predictive completion, but don't like the way it works "out-of-the-box" the rest of this manual explains all the options and features in detail.

Use the `predictive-mode` command to switch the predictive minor-mode on. The same command will also switch it off again. This section only covers (some of) the *default* behaviour of `predictive-mode`. The way that `predictive-mode` works can be radically changed via numerous customization options, which are described in detail in the rest of this manual.

As you add characters to a word, predictive mode searches in a dictionary for words starting with those characters. There are a number of different ways to choose which word (if any) should be used to complete what you've typed.

The most likely completion is provisionally inserted in the buffer after the point, and highlighted to indicate that it has not yet been accepted. The most likely completion is updated as you add more characters to the word. Use *C-RET* to accept a completion. See Section 4.4 [Dynamic Completion], page 12, for details.

What if you don't want to use the most likely completion? As you type, a list of the ten most likely completions is displayed in the echo area and, after a short delay, in a tooltip. They are ranked according to likelihood. Typing a number *0*–*9* will insert the corresponding completion. See Section 4.5 [Completion Hotkeys], page 13, Section 4.6 [Displaying Completions in the Echo Area], page 13, and Section 4.7 [Completion Tooltip], page 13 for details.

You can also display the completion tooltip manually, using *S-down*. When the tooltip is displayed, the *up* and *down* keys can be used to select a completion from the list in tooltip.

You can cycle forwards and backwards through the available completions even when the tooltip isn't displayed, using *M-TAB* or *M-/* and *M-SHIFT-TAB* or *M-?*. As you cycle, the next completion is provisionally inserted after the point and highlighted. *C-RET* accepts the completion, as usual.

Hitting *TAB* whilst completing will perform traditional "tab-completion" on the word: the word will be completed up to the longest common prefix of the available completions.

Another useful command is *C-TAB*. This accepts all the characters from the current provisional completion as though you typed them yourself, and re-completes the resulting, longer string — useful if a word just needs a suffix adding, for instance.

See Section 4.2 [Basic Completion Commands], page 9, for details of cycling and tab-completion.

M-Down will display a menu from which you can select a completion (see Section 4.9 [Completion Menu and Browser], page 15). There is also a '`Browser`' menu item, which brings up the completion browser. This hierarchically lists *all* possible completions, not just the most likely ones. (It can sometimes take a while to construct the completion browser, but you can hit *C-g* at any time to cancel it). See Section 4.9 [Completion Menu and Browser], page 15, for details.

`C-Down` displays a pop-up frame, in which you can use the usual Emacs motion keys to select a completion. `M-Tab` or `M-/` in the pop-up frame toggles between displaying the most likely completions and displaying all possible completions. To get rid of the pop-up frame, use `C-Up`.

The predictive completion package comes with an English dictionary, which is used by default. However, this is only really included to make sure everything works "out of the box". This default dictionary has already been trained on a large body of English text, which means it will take a very long time to adapt to your individual writing style. Don't expect predictive completion mode to display particularly good predictive powers if you stick to the default dictionary! For much better results, you should create your own dictionary (based on the supplied one if you like), and train it on samples of your own writing. See Chapter 5 [Dictionaries], page 20.

4 Completing Words

The minor-mode command `predictive-mode` toggles the mode on and off. With a positive prefix argument it switches the mode on, whilst a negative prefix argument turns it off and a zero prefix argument toggles. The `turn-on-predictive-mode` is also provided as a convenience for use in hooks.

4.1 Overview

Predictive mode is extremely flexible, and how you interact with it depends heavily on how you customize it. Broadly, there are two and a half ways to use predictive mode: it can either do nothing until you ask it to complete a word, or it can automatically search for the most likely completions as you type. (You may have seen something similar to the latter on certain models of mobile phone, though predictive mode is far more powerful.) This behaviour is controlled by enabling or disabling `auto-completion-mode` (see Section 4.3 [Auto-Completion Mode], page 11). It is enabled for you automatically if the `predictive-auto-complete` customization option is enabled (the default).

The half-a-way to use predictive mode is to set `predictive-auto-correction-no-completion`. In that case, predictive mode doesn't complete words at all, it only auto-corrects words you typed. For this to be useful, you also have to customize a number of options appropriately. See Section 4.11 [Miscellaneous Options], page 16.

Even if you are using `auto-completion-mode`, you can (by default) continue to type normally. Possible completion candidates will be displayed, ranked in order of likelihood, but will not be accepted unless you do so explicitly using `completion-accept`, bound to *`C-RET`*. (All the key bindings described here are the defaults, and can of course be customized to anything you desire, see Section 6.1.1 [Keymaps and Key Bindings], page 32) However, since it learns as you type, predictive mode can become very good at predicting the word you want, and it can become tedious to have to hit *`C-RET`* all the time. In that case, you can customize `auto-completion-syntax-alist` so that typing a punctuation or white-space character automatically accepts the current completion, before inserting the character. This lets you skip typing the rest of the characters in a word as soon as predictive mode has found the completion you want. However, the down side is that you have to slightly change the way you type, and it can take a little while until your fingers "get used to it". If you need to reject a completion candidate, you can do so using *`C-DEL`* or *`C-SPC`* (*`DEL`* is usually the *`backspace`* key).

If you aren't using `auto-completion-mode`, you will need to call the `complete-predictive` command whenever you want to complete the word at or next to the point. This is bound to *`M-TAB`*, *`M-SHIFT-TAB`*, *`M-/`* and *`M-?`*. There are then two "styles" of behaviour: completions can either be ephemeral, acting purely as a visual indicator, and disappearing unless you explicitly accept them using `completion-accept`, bound to *`C-RET`*. (You may be used to this style of behaviour from word processing software such as OpenOffice Writer). Alternatively, completions can be permanent, so that completing a word really does insert those characters into the buffer unless you explicitly reject it using `completion-reject`, bound to *`C-DEL`*. (This is the more traditional Emacs-style behaviour, that you may be used to from using `dabbrevs`, for example.) The choice is yours, and is controlled by the `completion-accept-or-reject-by-default` customization option (see Section 4.11 [Miscellaneous Options], page 16). (Note that these two different "styles" of

behaviour are mainly relevant when Dynamic Completion is enabled, as it is by default. This is described below.)

Whether you are using `auto-completion-mode` or not, there are various ways to display and select completions. All of them can be enabled or disabled indepentently, and many of them can be extensively customized.

Perhaps the simplest is to display the most likely completion in the buffer (see Section 4.4 [Dynamic Completion], page 12). You can then cycle through the other completion candidates using the `completion-cycle` command. *M-TAB* and *M-/* cycle forwards, whereas *M-SHIFT-TAB* and *M-?* cycle backwards. (These are the same key bindings used to manually complete the word at the point. Hitting them for the first time will bring up the possible completions, hitting them again will cycle.)

You can also insert a completion candidate directly, without having to cycle through them until you get to the one you want, by using a completion hot-key (see Section 4.5 [Completion Hotkeys], page 13). By default, the hot-keys are the number keys *0*–*9*, which insert the first through to the tenth candidate, respectively.

Of course, you won't know which completion candidate you want unless you can see which candidates are available! The completion candidates can be displayed in the echo area, and/or in a tooltip below the point (see Section 4.6 [Displaying Completions in the Echo Area], page 13, and see Section 4.7 [Completion Tooltip], page 13). When completion hot-keys are enabled, both of these also indicate which completion candidate corresponds to which hot-key. If you only want the hotkeys to be enabled when the tooltip or pop-up frame (see below) are displayed, then set `completion-use-hotkeys` to `pop-up`.

The completion tooltip is more than just a visual reminder of which completions are available. When it is displayed, you can select a completion from the list using the *up* and *down* arrow keys. When completing a word, you can display the tooltip at any time using `completion-show-tooltip`, bound to *S-down*. See Section 4.7 [Completion Tooltip], page 13.

A slightly more powerful alternative to the completion tooltip is a pop-up frame, displayed using `completion-popup-frame`, bound to *C-down* (see Section 4.8 [Pop-Up Frame], page 14). This lists the completion candidates in a separate Emacs frame positioned below the point, and you can use the usual Emacs motion keys to move through the list and select a candidate. However, you can also toggle between displaying just the most likely completion candidates and displaying all possible completions, using `completion-popup-frame-toggle-all`, bound to *M-Tab* and *M-/*. You can still type and delete characters when the pop-up frame is displayed; it will be updated to reflect the new set of completion candidates. To get rid of a pop-up frame, use `completion-popup-frame-dismiss`, bound to *C-up* and *M-up*. (Note that the pop-up frame key bindings are only active when the pop-up frame has the focus. If you manually switch the focus back to the original frame, you can still type normally and the pop-up frame will be updated appropriately, but the `completion-popup-frame-toggle-all` and `completion-popup-frame-dismiss` bindings will not work.)

The completion menu is the most flexible way of selecting completions, and can be displayed using `completion-show-menu`, bound to *M-down*. Again, it displays the completion candidates and lets you select them, and, like the pop-up frame, it also allows you to see all possible completions, rather than just displaying the most likely ones, by selecting the

'`Browser`' item from the menu. The completion browser doesn't just display all possible completions in a big list. Instead, it organises them hierarchically, making it easier to browse through them until you find the one you want.

The completion tooltip, pop-up frame, and menu all display a list of completion candidates, but each has its own advantages and disadvantages. The tooltip is visually and functionally least intrusive, but also gives you fewer ways to find the completion you want. The pop-up frame is encumbered by your window manager's frame decoration and positioning policy (though some window managers allow you to disable this for specific windows – called frames in Emacs). But it can display large numbers of completion candidates far more effectively than a tooltip. The completion menu and completion browser make finding the desired completion much easier, especially when you're not sure precisely what it is you're looking for. But menus steal the keyboard focus in Emacs, so you can no longer type in the buffer until you dismiss the menu.

You can select one of these three (tooltip, pop-up frame or menu) to be displayed automatically when you complete a word, by setting the `completion-auto-show` customization option. You can optionally leave a short delay before it is displayed, by setting `completion-auto-show-delay`[1].

4.2 Basic Completion Commands

4.2.1 Inserting Completions

`complete-predictive`
: (*`M-TAB`*, *`M-SHIFT-TAB`*, *`M-/`*, *`M-?`*) Find completions for the word at or next to the point.

`completion-cycle`
: (*`M-TAB`*, *`M-SHIFT-TAB`*, *`M-/`*, *`M-?`*) Cycle through available completion candidates. When supplied with a prefix argument, it will jump that many completions forwards (or backwards if the prefix argument is negative). *`M-/`* and *`M-?`* cycle backwards (so a negative argument will cause them to jump *forwards* that number of completions).

`completion-accept`
: (*`C-RET`*) Accept the current completion, and move the point just beyond it. If `predictive-auto-learn` and/or `predictive-auto-add` are enabled, predictive mode will also learn the word (see Section 5.5 [Dictionary Learning], page 25).

`completion-reject`
: (*`C-Backspace`*, *`C-SPC`*) Abandon the current completion, removing the provisionally inserted (i.e. highlighted) characters. If a prefix argument is supplied, predictive mode will also learn the resulting word, i.e. the prefix with the provisional completion removed (see Section 5.5 [Dictionary Learning], page 25). The *`C-SPC`* binding is only active in `auto-completion-mode` (see Section 4.3 [Auto-Completion Mode], page 11).

[1] Although you can choose to display the completion menu automatically, because it steals the keyboard focus this probably isn't all that useful, at least not without a long delay.

`completion-tab-complete`
: (*TAB*) Do "traditional" tab-completion, i.e. insert the longest common prefix of all candidate completions, and re-complete the resulting longer string.

`completion-extend-prefix`
: (*C-TAB*) Insert the characters from the current completion as though typed manually, and look for completions of the resulting, longer string.

Note that, apart from the `complete-predictive` bindings, none of the other key bindings are active until you have started completing a word, either using `complete-predictive` or automatically if you are using `auto-completion-mode` (see Section 4.3 [Auto-Completion Mode], page 11).

4.2.2 Deleting Characters

`completion-delete-char`
: (*<delete>*) Delete forwards and, after deleting, reject any completion at the point. A prefix argument sets the number of characters to delete.

`completion-backward-delete-char`
: Reject current completion, if there is one, and delete backwards. A prefix argument sets the number of characters to delete. If this deletes into a word and `auto-completion-mode` is enabled, complete what remains of that word.

`completion-backward-delete-char-untabify`
: (*DEL*) Similar to `completion-backward-delete-char`, but changes tabs to spaces as it deletes.

`completion-kill-word` (*C-<delete>*)
`completion-kill-sentence`
`completion-kill-sexp`
`completion-kill-paragraph`
: Similar to `completion-delete-char`, but kill forward until the end of the word/sentence/sexp/paragraph, instead of deleting individual characters.

`completion-backward-kill-word` (*C-DEL*)
`completion-backward-kill-sentence`
`completion-backward-kill-sexp`
`completion-backward-kill-paragraph`
: Similar to `completion-backward-delete-char`, but kill backward until the end of the word/sentence/sexp/paragraph, instead of deleting individual characters.

These commands replace the standard Emacs deletion and kill commands, so that they deal sensibly with any provisional completion that might be encountered in the region being deleted. If `auto-completion-mode` is enabled, the backwards deletion commands also re-complete any remaining prefix when they delete backwards into part of a word (see Section 4.3 [Auto-Completion Mode], page 11). In all other respects, these commands are identical to the equivalent standard Emacs deletion commands with corresponding names.

4.3 Auto-Completion Mode

When the `auto-completion` minor mode is enabled by calling the `auto-completion-mode` command, predictive mode will automatically look for completions of words as you type. This is especially useful when used in conjunction with dynamic completion (see Section 4.4 [Dynamic Completion], page 12). If you would like `auto-completion-mode` to be enabled automatically whenever predictive mode is enabled, set the `predictive-auto-complete` variable (set by default). Otherwise, you will not only have to enable `auto-completion-mode` manually, but also manually set the `auto-completion-source` to the `predictive` setting.

`auto-completion-mode`
: Enable auto-completion Mode. In this minor mode, completions are found automatically as words are typed.

`predictive-auto-complete`
: When non-nil (the default), enabling and disabling predictive mode will also automatically enable or disable `auto-completion-mode`, without needing to run the `auto-completion-mode` command manually.

`auto-completion-source`
: If `predictive-auto-complete` is *not* enabled, then you must set this variable manually to the `predictive` setting in order to use predictive auto-completion.

Word-constituent characters are added to the current word before finding new completions. The current syntax table (see Section "Syntax" in *GNU Emacs Manual*) determines which characters are word-constituents and which are not (although the behaviour of individual characters can be overridden).

By customizing `auto-completion-syntax-alist`, you can make punctuation and whitespace characters automatically accept the current completion. If you want to exceptionally insert a punctuation character as though it were a word-constituent (such as the '`.`' in '`e.g.`'), you can often prefix the character key with *M-*. The bindings *M-.* and *M--* are already set up by default, but you can easily add more using the `completion-define-word-constituent-binding` convenience function in your '`.emacs`' file. See Section 6.1 [Character Syntax and Key Bindings], page 32, for more details about the syntax and key binding features.

The following variables affect the behaviour of `auto-completion-mode`:

`auto-completion-syntax-alist`
: Whether `auto-completion-mode` allows you to type normally, ignoring the provisional completions until you want to use one (select '`type normally`' in the customization buffer), or whether punctuation characters automatically accept the current provisional completion before the character is inserted (select '`punctuation accepts`' in the customization buffer). For full details, see Section 6.1 [Character Syntax and Key Bindings], page 32.

`auto-completion-min-chars`
: Minimum number of characters that must be typed before the various completion mechanisms are activated when using `auto-completion-mode`. Note that you can still explicitly invoke completion on a shorter prefix using the

`complete-predictive` command (see Section 4.2.1 [Inserting Completions], page 9).

`auto-completion-delay`
: Number of seconds Emacs must be idle before the various completion mechanisms are activated when using `auto-completion-mode` (see Section 4.3 [Auto-Completion Mode], page 11).

`auto-completion-backward-delete-delay`
: Number of seconds Emacs must be idle after a backwards deletion (see Section 4.2.2 [Deleting Characters], page 10) before the various completion mechanisms are activated when using `auto-completion-mode` (see Section 4.3 [Auto-Completion Mode], page 11). Having a small delay is useful when the delete key is held down continuously. Default is 0.1 seconds.

4.4 Dynamic Completion

Dynamic completion is controlled by the `completion-use-dynamic` customisation variable. When it is non-nil, the most likely completion is provisionally inserted in the buffer after the point and highlighted. If `auto-completion-mode` is enabled, typing more word-constituent characters will add them to the word, updating the most likely completion (see Section 4.3 [Auto-Completion Mode], page 11).

Note that since the `completion-reject` command ends the completion process, if you want to find completions for the same prefix again after previously rejectng a completion, you have to do it manually with `complete-predictive` even when `auto-completion-mode` is enabled. See Section 4.2.1 [Inserting Completions], page 9, and Section 4.3 [Auto-Completion Mode], page 11.

`completion-use-dynamic`
: When non-nil, enable dynamic completion, which provisionally inserts the most likely conpletion in the buffer.

`completion-dynamic-highlight-common-substring`
: When non-nil, the longest common substring of all the available completions is highlighted in a different colour within the dynamic completion.

`completion-dynamic-highlight-prefix-alterations`
: When non-nil, any differences between the prefix you typed and the corresponding characters from the current completion are highlighted in a different colour.

`completion-dynamic-common-substring-face`
: The face used to highlight the longest common substring in a dynamic completion.

`completion-dynamic-prefix-alterations-face`
: The face used to highlight differences between the typed prefix and the corresponding characters in the current completion.

4.5 Completion Hotkeys

When the customisation variable `completion-use-hotkeys` is non-nil, you will be able to select a completion by typing a single character (the numerical characters *0*–*9* are used by default).

You will probably also want to display a list of the possible completions, so that you know which completion will be selected by each of these "hotkeys" (see Section 4.6 [Displaying Completions in the Echo Area], page 13, and Section 4.7 [Completion Tooltip], page 13). By setting `completion-use-hotkeys` to `pop-up`, the hotkeys will only be enabled when the tooltip or pop-up frame (see Section 4.8 [Pop-Up Frame], page 14) are displayed.

The following variables affect the behaviour of the completion hotkeys:

`completion-use-hotkeys`
: When non-nil, enable completion hotkeys, allowing completion candidates to be selected by hitting a single key. When set to the symbol `pop-up`, the hotkeys are only enabled when the tooltip (see Section 4.7 [Completion Tooltip], page 13) or pop-up frame (see Section 4.8 [Pop-Up Frame], page 14) are displayed.

`completion-hotkey-list`
: List of keys to use for selecting completions. Default is numerical characters 0 to 9. Note that this variable must be set *before* loading the predictive mode libraries, e.g. in your '`.emacs`' file. See Section 6.1.1 [Keymaps and Key Bindings], page 32.

Of course, however many characters are in `completion-hotkey-list`, there cannot be more completions available than the number actually found! The maximum number to find is limited by `completion-max-candidates`, Section 4.11 [Miscellaneous Options], page 16.

4.6 Displaying Completions in the Echo Area

If the customisation variable `completion-use-echo` is non-nil, a list of completion candidates is displayed in the echo area. If `completion-use-hotkeys` is also enabled (see Section 4.5 [Completion Hotkeys], page 13, the hot-key characters will be displayed next to the completions they select.

`completion-use-help-echo`
: When non-nil, a list of completion candidates (along with the hotkeys that select them, if enabled) is displayed in the echo area.

4.7 Completion Tooltip

Calling `completion-show-tooltip` when completing, bound to *S-Down* when `completion-use-tooltip` is enabled, displays a list of available completions in a tooltip. The `completion-tooltip-cycle` and `completion-tooltip-cycle-backwards` commands, bound to the *down* and *up* arrow keys, can then be used to select a completion from the list. The following variables affect the completion tooltip:

The tooltip can also be displayed automatically when completing, optionally after a time-delay. See Section 4.10 [Auto-Show a List of Completions], page 15.

`completion-tooltip-timeout`
: The number of seconds to display the tooltip. The tooltip is hidden automatically when it is no longer needed, but it is impossible to display a tooltip indefinitely in Emacs. The work-around is to set this to a very large value. Default is 86400.

`completion-tooltip-offset`
: A cons cell containing the number of pixels (an integer) by which to offset the tooltip by in the x and y directions (car and cdr, respectively).

`completion-tooltip-face`
: The face to use in the tooltip. Only the `:foreground`, `:background` and `:family` attributes are actually used.

4.8 Pop-Up Frame

Calling `completion-popup-frame` when completing, bound to *C-Down*, displays the available completions in a separate Emacs frame positioned below the point. You can then use the usual Emacs motion keys to select a completion candidate and provisionally insert it in the buffer (*up*, *down*, *M-p*, *M-n*, *C-p*, *C-n*, *pageup*, *C-v*, *M-v*, *pagedown*, *home*, *M-<*, *end* and *M->*; you can also supply numerical optional arguments to these, as usual).

You can also toggle between displaying just the most likely completions or displaying all possible completions using `completion-popup-toggle-all`, bound to *M-tab* and *M-/*. If there are a lot of possible completions, it can take a while before they're displayed. If it's taking too long, use *C-g* to cancel. To dismiss the pop-up frame, use `completion-popup-frame-dismiss`, bound to *C-up* and *M-up*.

The pop-up frame can also be displayed automatically when completing, optionally after a time-delay. See Section 4.10 [Auto-Show a List of Completions], page 15.

`completion-popup-frame`
: (*C-down*) Display completion candidates in a pop-up frame.

`completion-popup-frame-toggle-all`
: (*M-tab*, *M-/*) Toggle between displaying the most likely completion candidates and displaying all possible completions.

`completion-popup-frame-dismiss`
: (*C-up*, *M-up*) Dismiss the pop-up frame.

The following variables affect the pop-up frame:

`completion-popup-frame-max-height`
: An integer specifying the maximum height (in rows) of pop-up frames.

`completion-popup-frame-offset`
: A cons cell containing the number of pixels (an integer) by which to offset the pop-up frame by in the x and y directions (car and cdr, respectively).

4.9 Completion Menu and Browser

Calling `completion-show-menu` when completing, bound to *M-Down*, will bring up the completion menu, from which you can select a completion to insert. If `completion-use-hotkeys` is enabled, the menu will also display the hotkeys next to the completions they select, although you will have to exit the menu before you can use them (see Section 4.5 [Completion Hotkeys], page 13).

The completion menu also contains a `Browser` entry, which replaces the completion menu with the completion browser. This hierarchically lists *all* possible completions, irrespective of the setting of `completion-max-candidates` (see Section 4.11 [Miscellaneous Options], page 16). If it's taking too long to construct the browser, you can hit *C-g* to cancel it. You can also display the browser directly using `completion-show-browser-menu` (not bound to any key by default).

The completion menu can be displayed automatically when completing, optionally after a time-delay, though because it steals the keyboard focus this is less useful than auto-displaying the tooltip or pop-up frame. See Section 4.10 [Auto-Show a List of Completions], page 15.

The following variables affect the behaviour of the completion menu and browser:

`completion-menu-offset`
: A cons cell containing the number of pixels (an integer) by which to offset the menu by in the x and y directions (car and cdr, respectively).

`completion-browser-max-items`
: Maximum number of completions to display in a completion browser menu. If there are more completions than this in a menu, the menu will be divided into submenus, and if necessary the submenus will be further divided into subsubmenus, and so on ad infinitum. Default is to 25.

`completion-browser-buckets`
: Chooses the algorithm used to subdivide browser menus into submenus. The symbol `balance` causes the number of entries in all menus, submenus, subsubmenus etc. to be made as equal as possible. The symbol `max` maximizes the number of entries in higher level menus (and hence minimizes the number of entries in lower level submenus), whereas `minimize` does the opposite. Note that none of these options affect the number of levels of submenu required in a given browser instance.

`completion-browser-buckets`
: When set, the completion browser will recursively list completions of completions (of completions of completions...), organised hierarchically. Otherwise, the browser will only display the original list of all completions of the prefix.

4.10 Auto-Show a List of Completions

One out of the completion tooltip, pop-up frame, or menu can be displayed automatically when you start completing (it makes no sense to display more than one of them at a time; they would simply mask each other). It can either be displayed immediately, or only after Emacs has been idle for a number of seconds. The following variables control this feature:

`completion-auto-show`
: When set to `tooltip`, `pop-up` or `menu`, the corresponding list of completions is displayed automatically when completing. When nil, nothing is displayed automatically (they can still be displayed manually when required).

`completion-auto-show-delay`
: Number of seconds (integer) that Emacs must be idle before the list of completions is displayed.

4.11 Miscellaneous Options

The following variables affect the overall behaviour of predictive mode:

`completion-accept-or-reject-by-default`
: Determines the default action for the current completion. The options are: `accept` and `reject`, which accept or reject the completion, and `accept-common`, which accepts the longest common substring of the completion but deletes the rest.

`completion-how-to-resolve-old-completions`
: Determines what to do with old, abandoned completions elsewhere in the buffer. The options are: `accept` and `reject`, which accept or reject old completions, `leave`, which just leaves any old completions in place to return to later if you so desire, and `ask`, which asks you whether you want to accept or reject each completion.

`completion-hot-to-resolve-old-completions` controls what happens when you move the point away from a provisional dynamic completion (see Section 4.4 [Dynamic Completion], page 12) and start typing elsewhere in the buffer. When `auto-completion-mode` is disabled, `completion-accept-or-reject-by-default` determines how the *current* completion behaves. If it is set to `reject`, the completion user-interfaces serve only as a visual indicators; a completion will not become part of the buffer unless you explicitly accept it. If `completion-accept-or-reject-by-default` is set to anything else, completions really are part of the buffer, and you must explicitly reject them to get rid of any inserted characters.

When `auto-completion-mode` is enabled, `completion-accept-or-reject-by-default` has no effect. More fine-grained control is instead provided by `auto-completion-syntax-alist` and `auto-completion-override-syntax-alist`. See Section 4.3 [Auto-Completion Mode], page 11.

There is one exception to all this: whether or not `auto-completion-mode` is enabled, if you move the point to somewhere *within* a dynamic completion and start typing, the part of the completion before the point is always accepted (and the remaining characters deleted). This is almost always what you intended, and leads to less surprises.

`completion-overwrite`
: When non-nil, completions overwrite the rest of the word after point, both when you manually call `complete-predictive` with the point positioned in the middle of a word, and when `auto-completion-mode` is enabled and you type a new character in the middle of a word. Enabled by default.

Predictive mode doesn't play all that well with `overwrite-mode`. The `completion-overwrite` option implements an intelligent, partial over-write behaviour for completions.

This only has an effect if you try to start completing with the point in the middle of a word. When `completion-overwrite` is enabled, the part of the word at point that comes after the point will be over-written by the completion. When disabled, the completion is simply inserted in the middle of the word, without deleting the rest of it.

`completion-max-candidates`
: Maximum number of completions to find. Default is 10.

Setting `completion-max-candidates` to a large number is probably not useful, and will slow predictive mode down. It is easier to type a few extra characters than cycle through lots of completions, and the number available directly via hotkeys is limited by the number of keys you are prepared to set aside for selecting completions (see Section 4.5 [Completion Hotkeys], page 13).

`completion-highlight-face`
: The face used to highlight the completion candidates in the various user-interfaces.

`completion-highlight-face` is used to highlight the current dynamic completion in the buffer, and also to highlight the currently selected completion in the tooltip and pop-up frame.

`predictive-auto-correction-no-completion`
: When non-nil, predictive mode won't complete words at all! Instead, it will only auto-correct the words you type, using the definitions in `predictive-equivalent-characters` and `predictive-prefix-expansions` (see below). This is only useful if one or both of those variables have been set.

Setting `predictive-auto-correction-no-completion` changes predictive mode from being a completion mode to being an auto-correction mode. It relies on you defining equivalent characters in `predictive-equivalent-characters` or useful prefix expansions in `predictive-prefix-expansions` (see below). For example, if the former defines all accented variants of characters to be equivalent, then predictive mode will auto-correct accents for you, but without offering completions of the words. If you enable `predictive-auto-correction-no-completion`, you will almost certainly want to enable `auto-completion-mode` (see Section 4.3 [Auto-Completion Mode], page 11, change the default `auto-completion-syntax-alist` '`Acceptance behaviour`' to '`punctuation accepts`' (see Section 6.1.2 [Syntax], page 33), and set `completion-accept-or-reject-by-default` to `accept` (see above).

`predictive-equivalent-characters`
: A list of characters to be treated as equivalent. Each element of the list should be a string, and all characters appearing in the same string will be treated as equivalent when completing words. Predictive mode will then not only find completions for the prefix you typed, but also for all equivalent prefixes. Note that case is significant.

`predictive-prefix-expansions`
: An alist of expansions to apply to a prefix before completing it. The alist should associate regexps with their replacements. The result of expanding a prefix should be a valid regexp (but see below), which is used to match prefixes that

should be considered equivalent for completion. The expansions are applied in-order to the completion prefix. Characters matching a regexp are only expanded once, i.e. later expansions are *not* applied to the replacement text of previous expansions. Case is always significant.

The result of expanding a prefix according to `predictive-equivalent-characters` and `predictive-prefix-expansions` must produce a valid regexp, which is used to match prefixes that are considered equivalent to the one actually typed. Only a subset of the full Emacs regular expression syntax is supported. There is no support for regexp constructs that are only meaningful for strings (character ranges and character classes inside character alternatives, and syntax-related backslash constructs). Back-references and non-greedy postfix operators are not supported, so '?' after a postfix operator loses its special meaning. Also, matches are always anchored, so '$' and '^' lose their special meanings (use '.*' at the beginning and end of the regexp to get an unanchored match).

`predictive-equivalent-characters` works by substituting a character alternative listing all the equivalent characters whenever those characters appear in the prefix. It merely provides a more convenient way of defining these commonly used expansions, and is exactly the same as adding those expansions on to the very *end* of `predictive-prefix-expansions`. Any expansions defined in `predictive-prefix-expansions` therefore take precedence over character equivalences defined in `predictive-equivalent-characters`.

The main use of `predictive-equivalent-characters` is to make certain characters, e.g. the same character with and without diacritics, equivalent as far as completion is concerned. For example, if `predictive-equivalent-characters` was set to

```
("[eéêè]" "[EÉÊÈ]" "[aâà]" "[AÂÀ]")
```

then all accented and unaccented versions of '`e`' will be treated as equivalent, and similarly for '`a`'. So typing '`et`' would offer '`être`' and '`était`' as completions, as well as '`et`'. In this way, predictive mode can automatically correct accents and other diacritics as you type words. As with any auto-correction or spell-checker, be careful when using this: if there are two words that are identical up to diacritics, such as '`a`' and '`à`', then predictive mode can't telepathically know which one you want[2], and will insert whichever is the most likely.

Pre-defined `predictive-equivalent-characters` and `predictive-prefix-expansions` settings for some languages can be selected when customizing these variables.

`predictive-ignore-initial-caps`
: Controls whether predictive mode should ignore initial capital letters when searching for completions. If non-nil (the default), completions for the uncapitalised string are also found.

When `predictive-ignore-initial-caps` is set, only the *first* capital letter of a string is ignored. Thus typing `A` would find '`and`' (which would complete to '`And`'), '`Alaska`' and '`ANSI`', but typing `AN` would only find '`ANSI`', whilst typing `a` would only find '`and`'.

`predictive-auxiliary-file-location`
: Controls where any auxiliary files generated by predictive mode should be saved. It can either be a relative path, or an absolute path, but the former is *strongly*

[2] Telepathy support is slated for inclusion in version 1.0

recommended. If it is a relative path, it is taken to be relative to the file that a predictive-mode buffer is visiting.

If an absolute path is used, all auxiliary files for all predictive-mode buffers will be saved to the same location. In this case, there are *no* safe-guards to prevent two different auxiliary files that happen to have the same name from clobbering one another. That said, only identically named files in different directories pose a risk.

Depending on the settings you have chosen, predictive mode may not create any auxiliary files at all. The only one created in standard predictive mode buffers is the buffer-local dictionary (see Section 5.5.2 [Automatic Learning], page 26). However, the predictive mode support for a number of major-modes makes extensive use of auxiliary files. See Section 6.2 [Major Modes], page 35.

5 Dictionaries

Predictive completion is only as good as the dictionary it uses. The dictionary doesn't only list the words themselves, it also ranks them according to how likely they are, so that predictive mode can offer the most likely completions first.

As you type, predictive mode learns which words you use more frequently, so that the predictions improve. It can automatically ensure certain words are always ranked higher than others (useful e.g. when one word is a prefix for another).

Predictive mode is not restricted to using one dictionary at a time; it can use many dictionaries in parallel, and can automatically switch dictionaries in different regions of text, the regions being defined by regular expressions.

And predictive mode attempts to do all of that faster than you type, so that your typing is not slowed down even when using very large dictionaries. (As soon as Emacs becomes sentient – surely not far off – it will probably go on strike through being forced to work too fast!)

5.1 Creating Dictionaries

Predictive mode dictionaries store words along with their associated weights, used to rank the words in order of likelihood. The weight is just an integer value, which can be thought of as the relative frequency of a word (relative to the other words in the dictionary). A dictionary can also store prefix relationships between words, See Section 5.5.3 [Relationships Between Words], page 28.

The following commands are used to manually create and modify dictionaries:

`predictive-create-dict`
: Create a new dictionary. The dictionary name is read from the mini-buffer. You can optionally supply a filename to associate with the dictionary. The dictionary will be saved to this file by default (just as a buffer is saved to its associated file). You may also supply a file containing a list of words with which to populate the new dictionary. The `predictive-completion-speed` and `predictive-dict-autosave` variables set the new dictionary's completion speed and autosave flag (see below).

`predictive-create-meta-dict`
: Create a new meta-dictionary. A meta-dictionary is a wrapper around two or more dictionaries that behaves as if it was a single, combined dictionary. The weight of a word is the sum of it's weights in the constituent dictionaries, and the prefix relationships from all constituent dictionaries are merged (see Section 5.5.3 [Relationships Between Words], page 28). Apart from supplying a list of constituent dictionaries, the other options are identical to those for `predictive-create-dict`.

`predictive-add-to-dict`
: Insert a word into a dictionary. The dictionary name and word are read from the mini-buffer (defaults to the word at the point). An optional prefix argument specifies the weight. If the word is not already in the dictionary, it will be added to it with that initial weight (or 0 if none is supplied). If the word is already

in the dictionary, its weight will be incremented by the weight value (or by 1 if none is supplied).

`predictive-remove-from-dict`
: Completely remove a word from a dictionary. The dictionary name and word are read from the mini-buffer (defaults to the word at the point).

`predictive-reset-weight`
: Reset the weight of a word in a dictionary to 0. The dictionary name and word are read from the mini-buffer. If no word is supplied, reset the weights of all words in the dictionary. If a prefix argument is supplied, reset weight(s) to that value, rather than 0.

`dictree-size`
: Display the number of words in a dictionary.

The file containing the list of words used to populate a dictionary has to conform to a specific format:

```
"word" weight [prefix-list]
```

Each line contains one word *word*, delimited by '""', followed by an integer *weight* which specifies the word's weight, separated by white-space from the word itself. Note that the 'words' in a dictionary do not have to be words in the usual sense. They can be arbitrary sequences of characters, including white-space and punctuation characters. The quote character '"' can be included in a word by escaping it: '\"'. Optionally, a list of words which are prefixes of *word* can be specified in *prefix-list* at the end of the line, again separated from the *weight* by white-space (see Section 5.5.3 [Relationships Between Words], page 28). If present, it should be of the form:

```
(:prefixes ("prefix1" "prefix2" ...))
```

Note that trailing whitespace on any line is *not* allowed.

The following variables set defaults for other dictionary properties. To change their values for a single dictionary, set the variable to the desired value before creating the dictionary, resetting the value afterwards.

`predictive-completion-speed`
: Sets the default completion speed of dictionaries created with `predictive-create-dict`. This is the desired upper limit on the time it takes to find completions. If it takes longer than this to find a particular completion, the results are cached so that they can be retrieved faster next time. Thus lower values result in faster completion, at the expense of dictionaries taking up more memory.

  Due to the efficient data structures used by the dictionaries, it is typically safe to set this quite low (the default is 0.1 seconds). Most completions will be found faster than this even on slow computers, and only a few of the very slowest will need to be cached.

`predictive-dict-autosave`
: Sets the default autosave property for dictionaries created with `predictive-create-dict`. If non-nil, modified dictionaries will automatically be saved when they are unloaded (either with the `predictive-dict-unload` command, or

when exiting emacs). If nil, any unsaved modifications will be lost unless the dictionary is saved manually. See Section 5.2 [Loading and Saving Dictionaries], page 22.

5.2 Loading and Saving Dictionaries

`predictive-load-dict`
: Load a dictionary by name, and add it to the list of dictionaries used by the current buffer. The dictionary will be included when learning from the buffer, see Section 5.5.1 [Learning from Buffers and Files], page 25, and if its autosave flag is set (see Section 5.1 [Creating Dictionaries], page 20), it will automatically be saved when the buffer is killed. The dictionary file must be in your `load-path`.

  You should never normally need to use this command interactively, since predictive mode loads and unloads dictionaries automatically, as needed.

`dictree-load`
: Load a dictionary from file. The name of the loaded dictionary is the same as the file name, with the extension removed. This will not add it to the list of dictionaries used by the current buffer (see `predictive-load-dict`, above).

`predictive-unload-dict`
: Remove a dictionary from the list of dictionaries used by the current buffer. If the dictionary is no longer used by any other buffer, this also unloads it from Emacs. In that case, if its autosave flag is set, the dictionary will be saved before being unloaded (see Section 5.1 [Creating Dictionaries], page 20), unless this is overridden by supplying a prefix argument.

  You should never normally need to use this command interactively, since predictive mode loads and unloads dictionaries automatically, as needed.

`dictree-unload`
: You probably don't want to do this! Unloading a dictionary that's still in use will cripple predictive mode, resulting in it spewing out incomprehensible Lisp errors. This command unconditionally unloads a dictionary. If the dictionary's autosave flag is set, this will also save it (see Section 5.1 [Creating Dictionaries], page 20), unless overridden by supplying a prefix argument.

`predictive-save-dict`
: Save a dictionary to its associated file. Prompt for a file name if there is none associated with the dictionary.

`predictive-write-dict`
: Write a dictionary to a file specified via the mini-buffer. This also associates the dictionary with that file. If a prefix argument is supplied, you will *not* be asked to confirm if over-writing an existing file.

`predictive-save-modified-dicts`
: Save all modified dictionaries that have a non-nil autosave flag. If a prefix argument is supplied, prompt for confirmation before saving each dictionary.

`predictive-dump-dict-to-buffer`
: Dump all words, weights and prefix relationships in the dictionary to a buffer, in the same format as that used to populate dictionaries (see Section 5.1 [Creating Dictionaries], page 20).

`predictive-dump-dict-to-file`
: Dump words, weights and prefix relationships to a text file rather than a buffer. If a prefix argument is supplied, you will *not* be asked to confirm if over-writing an existing file.

`predictive-dict-autosave`
: Sets the default autosave property for dictionaries created with `predictive-create-dict`. If non-nil, modified dictionaries will automatically be saved when they are unloaded (either with the `predictive-dict-unload` command, or when exiting emacs). If nil, any unsaved modifications will be lost unless the dictionary is saved manually. See Section 5.2 [Loading and Saving Dictionaries], page 22.

`predictive-dict-compilation`
: Determines whether dictionaries are saved in compiled or uncompiled form, or both. If set to the symbol `compiled`, dictionaries are saved in compiled form, if set to `uncompiled` they are saved in uncompiled form, and if set to anything else they are saved in both forms (the default). A compiled dictionary can be loaded a lot faster, and is always used in preference to the uncompiled form if it exists. However, compiled dictionaries are not portable between different Emacs versions, whereas uncompiled ones are.

`predictive-dict-lock-loaded-list`
: List of dictionaries that should never be automatically unloaded, in addition to `predictive-main-dict` (see Section 5.3 [Basic Dictionary Usage], page 24).

To use a dictionary, it must be loaded into memory. Usually, predictive mode loads the dictionaries it needs automatically, and unloads them again when they are no longer needed. Once it has been loaded, the `predictive-main-dict` is never automatically unloaded. It can take a while to load large dictionaries, and `predictive-main-dict` is likely to be used by most predictive-mode buffers. If you would like to prevent any other dictionaries from being unloaded automatically, add them to `predictive-dict-lock-loaded`.

You can also load and unload dictionaries manually. If a dictionary is saved somewhere in your load path, you can load it using the `predictive-load-dict` and `dictree-load` commands. The `predictive-load-dict` command additionally adds it to the list of dictionaries used by the current buffer. This has two implications: the dictionary will automatically be saved when the buffer is killed (assuming its autosave flag is non-nil, see Section 5.1 [Creating Dictionaries], page 20), and it will be included when learning from the buffer (see Section 5.5.1 [Learning from Buffers and Files], page 25). (You should avoid loading a dictionary using the `load-file` command, as it will not necessarily ensure that the dictionary is correctly associated with the file it was loaded from).

If you want a dictionary to be loaded every time you run Emacs, and the dictionary is saved somewhere in your `load-path`, you can add the following line to your '`.emacs`' file:

```
(dictree-load 'dictionary-name)
```

The major-mode setup functions load the dictionaries they need automatically. See Section 6.2 [Major Modes], page 35.

A buffer is usually associated with a file, and saving the buffer with the Emacs `save-buffer` command writes any changes back to that file. Similarly, dictionaries are usually associated with a dictionary file. The `predictive-save-dict` command saves any changes back to that file. The `predictive-write-dict` command is analogous to the Emacs `write-file` command.

You should *never* rename a dictionary file. (Moving a dictionary file to another directory in your `load-path` is fine, but you cannot rename the file itself.) The correct way to rename a dictionary is to supply a new file name to `predictive-write-dict`.

Dictionaries can be modified by adding words to them with the `predictive-add-to-dict` command (see Section 5.1 [Creating Dictionaries], page 20). If the auto-learn features are used, dictionaries are modified whenever a completion is accepted (see Section 5.5.2 [Automatic Learning], page 26). The `predictive-save-modified-dicts` saves all modified dictionaries whose autosave flag is set. Modified dictionaries used in a buffer are automatically saved when that buffer is killed if they have their autosave flag set. All modified autosave dictionaries are saved when you exit Emacs.

5.3 Basic Dictionary Usage

The name of the main dictionary used by a buffer is stored in the buffer-local `predictive-main-dict` variable. Note that the variable usually contains the *name* of the dictionary (a symbol), not the dictionary itself. This is the dictionary predictive mode will normally search in when looking for completions.

`predictive-main-dict` can also hold a list of dictionary names. They are then treated as though they form one combined dictionary. However, when `predictive-auto-add-to-dict` is used, words are always added to the first dictionary in the list. See Section 5.5.2 [Automatic Learning], page 26.

`predictive-set-main-dict`
: This function can be used as a convenience to set the main dictionary for the current buffer. You will only be able to select dictionaries that are already loaded (see Section 5.2 [Loading and Saving Dictionaries], page 22). If you want to set the default main dictionary permanently, customize `predictive-main-dict` instead.

5.4 Region-Local Dictionaries

Using overlays, it is possible to set up dictionaries that are local to specific regions of text. The predictive mode support for a number of major modes makes use of this feature (see Section 6.2 [Major Modes], page 35). Configuring region-local dictionaries is, however, beyond the scope of this user manual. See Section "Region-Local Dictionaries" in *Predictive Programmer Manual*.

`predictive-which-dict-mode`
: Auxilliary minor-mode that displays the currently active dictionary in the mode-line.

Predictive mode provides an auxilliary minor-mode, `predictive-which-dict-mode`, which displays the name of the currently active dictionary in the mode-line, i.e. the dictionary that is active at the point. If the active dictionary is in fact a list of dictionaries, the name of the first dictionary in the list is displayed, followed by '`...`'. Positioning the mouse over the dictionary name displays the names of the other dictionaries, either in the echo-area (if `tooltip-mode` is disabled, or in a terminal) or in a tooltip. However, for this to work, you have to customize part of the `mode-line-modes` variable, as the default setting clobbers this functionality. Find the part that refers to the `minor-mode-alist` variable, and delete the `help-echo` property, i.e. change that part to:

```
(:propertize ("" minor-mode-alist)
 mouse-face mode-line-highlight local-map
 (keymap (header-line keymap (down-mouse-3 . mode-line-mode-menu-1))
         (mode-line keymap (down-mouse-3 . mode-line-mode-menu-1)
                           (mouse-2 . mode-line-minor-mode-help)
                           (down-mouse-1 . mouse-minor-mode-menu))))
```

5.5 Dictionary Learning

The better the weights in a dictionary match the frequency with which you use words, the more useful predictive mode will be. Some of the standard dictionaries already include word weights, which match average word frequencies taken from a large sample of texts. Some don't include any word weights. In any case, your personal word usage might be very different from the average.

Ideally, the weights in a dictionary should match your personal style of writing. In fact, since your writing style might change significantly depending on whether you're writing, say, a scientific article or an email, it may even be worth having different dictionaries for different circumstances.

The easiest way to teach a dictionary about your writing style is to supply it with samples of your writing, and have it learn the word weights from them. Predictive mode provides two ways to do this: learning from existing files, and automatic learning as you type.

5.5.1 Learning from Buffers and Files

Predictive mode can learn word weights from existing text. The following commands can be used to do this. Note that they will not add new words to a dictionary; they only update weights of words that are already in the dictionary. The learning commands take account of region-local dictionaries, and will ensure words are learnt in the correct dictionaries (see Section 5.4 [Region-Local Dictionaries], page 24).

Usually, you want to accumulate knowledge from each new piece of text. But sometimes, you may want to start from scratch, and reset the word weights to zero before starting to train a dictionary (for example, to erase the predefined weights from the supplied English dictionary, so that you can train it on your own writing style). You can use the `predictive-reset-weight` command for this purpose (see Section 5.1 [Creating Dictionaries], page 20).

Note that all the learning commands (even the "fast" ones!) can take a long time to run.

`predictive-learn-from-buffer`
: Learns weights for words in a dictionary from text in a buffer. If no explicit dictionary is specified, this learns word weights for all dictionaries used by the current buffer (see Section 5.2 [Loading and Saving Dictionaries], page 22). Each occurrence of a word increments its weight in the dictionary. By default, only occurrences that occur in a region where the dictionary is active are counted (see Section 5.4 [Region-Local Dictionaries], page 24). If an explicit dictionary is specified, this can be overridden by supplying a prefix argument, in which case all occurrences are counted, irrespective of whether the dictionary is active at the word occurrence. Note that you *cannot* use this command to add words to a dictionary, only to train the weights of words already in a dictionary (see `predictive-fast-learn-or-add-from-buffer`, below).

`predictive-learn-from-file`
: Like `predictive-learn-from-buffer`, but learns from a file instead of a buffer.

`predictive-fast-learn-or-add-from-buffer`
: Similar to `predictive-learn-from-buffer`. It runs faster for large dictionaries, at the expense of missing some words. Specifically, only words consisting entirely of word- or symbol-constituent characters (according to the buffer's syntax table) will be taken into account. Also, unlike `predictive-learn-from-buffer`, this command takes into account the setting of the `predictive-auto-add-to-dict` option. If an explicit dictionary is specified, words that are not already in the dictionary will be added to it if `predictive-auto-add-to-dict` has any non-nil value. If no explicit dictionary is specified, `predictive-auto-add-to-dict` has the usual effect, as do the other auto-add-related options (see Section 5.5.2 [Automatic Learning], page 26).

`predictive-fast-learn-or-add-from-file`
: Like `predictive-fast-learn-or-add-from-buffer`, but learns from a file instead of a buffer.

5.5.2 Automatic Learning

Predictive mode can automatically learn which words you use most often as you type, in order to make better predictions. This feature is especially useful when you first start using a dictionary, to adapt it to your writing style. Once a dictionary has been trained and is making good predictions, it can be turned off to fix the order in which completions are offered (see Chapter 1 [What is predictive completion?], page 1), though leaving it on usually doesn't cause the order to change that much.

The following variables control automatic learning:

`predictive-auto-learn`
: Controls automatic word frequency learning. When non-nil (the default), the weight for a word in is incremented each time it is accepted as a completion, making the word more likely to be offered higher up the list of completions in the future. Words that are not already in the dictionary are ignored unless `predictive-auto-add-to-dict` is set.

`predictive-auto-add-to-dict`
: Controls automatic adding of new words to dictionaries. If nil (the default), new words are never automatically added to a dictionary. If `t`, new words are automatically added to the active dictionary. If set to a dictionary name, new words are automatically added to that dictionary instead of the active one.

`predictive-add-to-dict-ask`
: If non-nil, predictive mode will ask for confirmation before automatically adding any word to a dictionary. Enabled by default. This has no effect unless `predictive-auto-add-to-dict` is also set.

`predictive-use-buffer-local-dict`
: If non-nil, a special, buffer-local dictionary will be created for each predictive mode buffer. The buffer-local dictionary is used in conjunction with the `predictive-main-dict`, and the two act as a single, combined main dictionary for the buffer.

  The buffer-local dictionary is initially empty, but whenever a word is learnt (auto-learnt, auto-added, learnt from a buffer or file, or added manually), it is added to the buffer-local dictionary, and its weight there is incremented by a value `predictive-buffer-local-learn-multiplier` times higher than for normal dictionaries. Thus the buffer-local dictionary will help predictive mode adapt much faster to the vocabulary used in a specific buffer than global dictionaries alone can.

  If `predictive-dict-autosave` is enabled and the buffer is associated with a file, the buffer-local dictionary will automatically be saved to the directory containing the file[1]. When you load the file in the future, predictive mode will look for the buffer-local dictionary in the same directory; there is no need to add the directory to your load path. If the buffer is not associated with a file, the buffer-local dictionary will be discarded when you end the Emacs session.

`predictive-buffer-local-learn-multiplier`
: Sets the learning speed for buffer-local dictionaries. Whenever a word is added to a buffer-local dictionary, the weight increment is multiplied by this value before being added to any existing word weight. The default is 50.

`predictive-use-auto-learn-cache`
: If non-nil (the default), auto-learnt and auto-added words are cached, and only actually added to the dictionary when Emacs has been idle for `predictive-flush-auto-learn-delay` seconds or the buffer is killed (it has no effect unless at least one of `predictive-auto-learn` or `predictive-auto-add-to-dict` is also set). This avoids small but sometimes noticeable delays when typing. New words or word weights will not be taken into account until the cache is fully flushed.

`predictive-auto-add-min-chars`
: Minimum length of words auto-added to the dictionary. When enabled, words shorter than this will be ignored when auto-add is used.

[1] In fact, two dictionaries will be saved in the directory, since the buffer-local dictionary is composed of a meta-dictionary and a normal dictionary, see Section 5.1 [Creating Dictionaries], page 20.

`predictive-auto-add-filter`
: When this variable is set to a function, and when `predictive-auto-add-to-dict` is enabled, the function will called whenever a word is going to be auto-added to the dictionary, passing the word (a string) and the dictionary as arguments. The word will only be added if the function returns non-nil. If `predictive-use-auto-learn-cache` is enabled, the filter function will be called when cached entries are flushed, not when they're added to the cache, allowing even time-consuming filter functions to be used.

Note that if `predictive-main-dict` contains a list of dictionary names (see Section 5.3 [Basic Dictionary Usage], page 24), an automatically learnt or added word may not end up where you want it. The weight of a word is incremented in the first dictionary it is found in, and words are added to the first dictionary in the list (assuming `predictive-auto-add-to-dict` is set to `t`). It is best to ensure that dictionaries in the list do not duplicate any words.

The `predictive-auto-add-filter` is not a customization option, so it can only be set from Lisp code (e.g. a setup function, see Section 6.2 [Major Modes], page 35). One example of its use would be to filter out words that contain non-letter characters (though it may be better to customize `completion-dynamic-syntax-alist` and `completion-dynamic-override-syntax-alist` instead, see Section 6.1.2 [Syntax], page 33). The following will accomplish this:

```
(setq predictive-auto-add-filter
      (lambda (word dict) (string-match "^[[:alpha:]]$" word)))
```

Note that the function must accept both the word and dictionary arguments, even if it doesn't make use of the dictionary.

Another example would be to check that words are spelled correctly before auto-adding them to a dictionary, either using `ispell` or using the English dictionary that comes with predictive mode. This sounds tautological, but it does make sense: the dictionary you use for predictive completion will only contain words you've used at least once, but typos and spelling mistakes won't make it into the dictionary (see Section 5.6 [Getting the Most out of Dictionaries], page 30).

```
(setq predictive-auto-add-filter (lambda (word dict) (lookup-words word)))
```

Using the supplied predictive English dictionary will be faster than `ispell`, since it is optimised for looking up words, though this isn't such an issue if `predictive-use-auto-learn-cache` is enabled (the following assumes `dict-english` is already loaded, see Section 5.2 [Loading and Saving Dictionaries], page 22).

```
(setq predictive-auto-add-filter
      (lambda (word) (dictree-member-p dict-english word)))
```

Other possible uses for `predictive-auto-add-filter` are limited only by your imagination!

5.5.3 Relationships Between Words

As well as word frequencies, predictive mode dictionaries can store certain relationships between words, to make learning more effective. With each word in a dictionary, a list of other words can be associated. Predictive mode automatically ensures that the weights of

the words in this list are always at least as large as that of the word they are associated with.

This is most useful when one word is a prefix for another. For example, you may want to ensure that, however frequently the word '`learning`' is used, the weight of the word '`learn`' is always kept at least as big, so that it always takes precedence when completing.

The following functions allow you to define and undefine such prefix relationships (note that despite the command names and descriptions, `predictive-define-prefix` and `predictive-undefine-prefix` can be used to define relationships between any two words, not just prefixes; however, `predictive-define-all-prefixes` and the `predictive-auto-define-prefixes` feature can only define actual prefix relationships):

`predictive-define-prefix`
: Define one word to be a prefix of another. Predictive mode will ensure that the weight of the "prefix" word is always at least as large as that of the other. (Note that the word does not actually have to be a prefix of the other; this can be used to define a relationship between any two words, so that the weight of one is always larger than the other.)

`predictive-undefine-prefix`
: Remove a "prefix" definition. (As for `predictive-define-prefix`, the word does not actually have to be a prefix of the other.)

`predictive-define-all-prefixes`
: Add the given prefix to the prefix definitions of all words that for which it is a prefix, or define all possible prefix relationships in the dictionary if no explicit prefix is supplied. In the latter case, a numerical prefix argument sets a minimum word length for which to define a prefix relationship; relationships are only defined for words that are at least this long (the *prefixes* defined for those words can still be any length).

`predictive-undefine-all-prefixes`
: The analogue of `predictive-define-all-prefixes`. Remove the given prefix from the prefix definitions of all words, or remove all prefix definitions in the dictionary if no explicit prefix is supplied. In the latter case, a numerical prefix argument sets a minimum word length for which to *undefine* prefix relationships (the length of the *prefixes* that are undefined is still not restricted in any way).

The following variable is used to help guess a likely prefix as a default for `predictive-define-prefix` and `predictive-undefine-prefix`. It's default value is only appropriate for English.

`predictive-guess-prefix-suffixes`
: List of suffixes to use when guessing a likely prefix for a word. The suffixes a tried in the order they appear in the list, and the first one that matches the end of the word is used: the guessed prefix is the original word with the suffix removed.

Ensuring that prefixes take precedence when completing words is almost always a good idea. It makes predictive completion much more convenient (especially dynamic completion, see Section 4.4 [Dynamic Completion], page 12). Therefore, predictive mode includes a

feature that, when enabled (the default), automatically defines all the prefix relationships whenever a word is added to a dictionary.

`predictive-auto-define-prefixes`
: When non-nil, predictive mode will automatically update all prefix relationships for a word in a dictionary when the word is added. The new word will always take precedence over any word that is an extension of it, and in their turn any words that are prefixes of the new word will take precedence over it.

5.6 Getting the Most out of Dictionaries

As it says at the beginning of this chapter, predictive completion is only as good as the dictionary it uses. The English dictionary supplied with the predictive package is trained on a large body of (British) English text, so the words and word weights it contains accurately reflect average English usage. But you are very unlikely to write "average" English (whatever that is!). To get the most out of predictive completion, it is better to train your dictionary on your own writing style, rather than someone else's.

There are two approaches to this. The first is to create a copy of the supplied English dictionary containing the all same words, but with all their weights reset to zero. You can then either use the auto-learn feature to slowly train the dictionary as you write (see Section 5.5.2 [Automatic Learning], page 26, or better still, kick-start things by training it on text you have already written by learning from existing files (see Section 5.5.1 [Learning from Buffers and Files], page 25). You can of course still leave auto-learn enabled in order to refine the dictionary, or even use the auto-add feature to automatically add missing words as you type them (see below).

A variant of this approach, if you don't like the supplied English dictionary, is to create the initial dictionary from some other list of words, e.g. the '`/usr/dict/words`' file on Unix systems. You will first need to massage the list into the format required by `predictive-create-dict` (see Section 5.1 [Creating Dictionaries], page 20), which is the same as the format produced by the dump commands (see Section 5.2 [Loading and Saving Dictionaries], page 22), but this should be easy for even a moderately savvy Emacs user[2]!

The second approach is to start from a completely empty dictionary, and use the auto-add feature to automatically add words as you type them (see Section 5.5.2 [Automatic Learning], page 26). The auto-add feature adds words when you "accept" them. Since the words aren't already in the dictionary, the easiest way to add new words while typing is to ensure dynamic completion is enabled, and type an end-of-word character (such as a space or punctuation character) at the end of the word (see Section 4.4 [Dynamic Completion], page 12). Alternatively, you can use the fast learning commands `predictive-fast-learn-or-add-from-buffer` and `predictive-fast-learn-or-add-from-file` to add words from existing text (note that you *must* use the fast learning commands for this; the normal ones will only increment the weights of words that are already in the dictionary).

However you auto-add the words, there is a risk that some words that you don't want will make their way into the dictionary, for example typos and misspellings, or possibly words containing non-letter characters. The latter are best dealt with by appropriate entries in `completion-dynamic-syntax-alist` and `completion-dynamic-override-syntax-alist` (see Section 6.1.2 [Syntax], page 33). The former are best dealt with by

[2] Keyboard macros may help here...

setting a `predictive-auto-add-filter` function (see Section 5.5.2 [Automatic Learning], page 26). It's still a good idea to occasionally check which words are in the dictionary by dumping it to a buffer and scanning through it by hand or with `ispell` (see Section 5.2 [Loading and Saving Dictionaries], page 22).

So which approach is better? Each has advantages and disadvantages, and it comes down to personal preference. Training a reset copy of the supplied English dictionary (or one built from another word list) ensures that all the words in the dictionary are spelled correctly (assuming the words in the list were correct in the first place). It also means that predictive mode will provide spelling assistance even when you type an obscure word that you've never used before. On the other hand, the dictionary will contain many words that you will never use, and may lack words that you do use, which will have to be added by hand (unless you enable auto-add).

If you write different types of text (e.g. your novel, academic papers, and emails), the vocabulary you use will differ significantly between the different types of text. You will get more out of predictive completion by creating separate dictionaries for each. You can then set up predictive mode to select the appropriate dictionary automatically, either based on the major mode (see Section 6.2 [Major Modes], page 35) or, in the case of LaTeX documents, based on the document class (see Section 6.2.1 [LaTeX Support], page 36).

Once you've created your dictionaries, you can use the many features of predictive mode to tweak the dictionary training and behaviour to suit your every desire. Using buffer-local dictionaries can help predictive mode adapt faster to the specific vocabulary you are using in an individual document, especially if you set a large `predictive-buffer-local-learn-multiplier` (see Section 5.5.2 [Automatic Learning], page 26). Defining sensible prefix relationships between words makes sure predictive completion doesn't "get in your way" when you're typing fast (see Section 5.5.3 [Relationships Between Words], page 28). The `predictive-auto-define-prefixes` option and the `predictive-define-all-prefixes` command make defining prefix relationships very easy.

Finally, having gone to all this effort to create the perfect dictionary, it would be tragic to lose it all! Make sure you occasionally backup your dictionaries by dumping them to a plain text file using `predictive-dump-dict-to-file` (see Section 5.2 [Loading and Saving Dictionaries], page 22). This is vital before upgrading to a new version of the Predictive package, since there's no guarantee that old dictionaries will be readable in the new version (whereas the dumped plain-text format is usually stable across Predictive package versions; even if exceptionally it changes, since it's a plain-text format it will at the very least always be readable in Emacs, and can be manipulated into the format required for recreating your dictionary in the new Predictive package version).

6 Advanced Customisation

This chapter describes the more advanced customisation features provided by predictive mode. Many of these are on the borderline between user-customisations and features to be used by Lisp packages that enhance predictive completion for specific major modes, though describing all the features available to Lisp packages is beyond the scope of this user manual. See Section "Region-Local Dictionaries" in *Predictive Programmer Manual*.

6.1 Character Syntax and Key Bindings

Predictive mode significantly changes what happens when normal, printable characters are typed. Different characters cause different behaviour. For example, letter characters will usually be added to the current word, updating the completions, whereas punctuation characters end the completion process.

This chapter describes the mechanisms that determine the behaviour of different characters, and how to customize them.

6.1.1 Keymaps and Key Bindings

The following keymaps are defined by predictive mode:

`predictive-map`
: Main keymap, enabled whenever predictive mode is.

`auto-completion-map`
: Keymap enabled whenever `auto-completion-mode` is.

`completion-hotkey-list`
: List of hotkey characters to use for selecting completions. Default is numerical characters 0 to 9.

`completion-overlay-map` `auto-completion-overlay-map`
: Keymaps active when the point is at or within a completion.

`completion-dynamic-map`
: Keymap used when `completion-use-dynamic` is enabled.

`completion-tooltip-map`
: Keymap used when `completion-use-tooltip` is enabled.

`completion-tooltip-active-map`
: Keymap active when the completion tooltip is displayed.

`completion-menu-map`
: Keymap used when `completion-use-menu` is enabled.

`completion-popup-frame-map`
: Keymap used when `completion-use-popup-frame` is enabled.

`completion-popup-frame-mode-map`
: Keymap active in the completion pop-up frame.

The various keymaps define key bindings for different situations that arise in predictive mode. The main `predictive-map` keymap is enabled whenever predictive mode is enabled,

and `auto-completion-map` is enabled whenever `auto-completion-mode` is. Note: if you find yourself thinking of re-binding printable characters in `auto-completion-map` to something other than `completion-self-insert`, don't! (at least not until you've read on a bit). What you probably want to change are the `auto-completion-syntax-alist` and `auto-completion-override-syntax-alist` variables. See Section 6.1.2 [Syntax], page 33.

`completion-overlay-map` and `auto-completion-overlay-map` are active when the point is located at a completion. The former is used when `auto-completion-mode` is disabled, the latter when it is enabled. They are used to provide key bindings for most of the completion features.

The key bindings in the keymaps enabled by the `completion-use-*` customization variables are effectively added to `completion-overlay-map`, so they are active when the point is in a completion and the corresponding user-interface is enabled.

If the keymaps are not defined when predictive mode is first loaded (see Chapter 2 [Obtaining and Installing], page 3), it creates the default keymaps and loads them into Emacs. Therefore, to completely re-define predictive mode key bindings, it is simpler to re-define the keymap variables *before* loading predictive mode (e.g. before the `(require 'predictive)` line in your '`.emacs`' file, see Chapter 2 [Obtaining and Installing], page 3). (However, doing this for the `auto-completion-mode-map` will almost certainly break `auto-completion-mode` entirely!) But if you simply want to modify a few key bindings, as will usually be the case, you can do it in the usual Emacs way, *after* loading predictive mode in your '`.emacs`' file.

6.1.2 Syntax

`auto-completion-syntax-alist`
: Alist associating character syntax descriptors with completion functions. Used by the `completion-self-insert` function in `auto-completion-mode` to decide what to do based on a typed character's syntax.

`auto-completion-override-syntax-alist`
: Alist associating characters with completion functions. Overrides the default function for a typed character's syntax. Used by `completion-self-insert` in `auto-completion-mode`.

`completion-define-word-constituent-binding`
: Convenience command for use in your '`.emacs`' file. It is used to define key bindings that insert a character as though it had a different syntax. Usually used to allow punctuation characters to be inserted one-off as word-constituents. To define *key* as a binding to insert character *char* as though its were a word-constituent, use:

```
(completion-define-word-constituent-binding key char)
```

To define *key* as a binding to insert character *char* as though its syntax class were *syntax*, use:

```
(completion-define-word-constituent-binding key char syntax)
```

To ignore `auto-completion-override-syntax-alist` for this key binding, so that the behaviour is determined only by *syntax*, supply a non-nil third argument:

```
(completion-define-word-constituent-binding key char syntax t)
```

When a character is typed in `auto-completion-mode`, predictive mode decides what to do based on that character's syntax, as defined by the current syntax table (see Section "Syntax" in *GNU Emacs Manual*). All printable characters are bound by default to the function `completion-self-insert`, which inserts the character, looks up the character's syntax descriptor in `auto-completion-syntax-alist`, and carries out the associated actions.

By default, all word-constituent characters (syntax descriptor `w`) insert the character and complete the new prefix, whilst anything else rejects any provisional completion and inserts the character.

Occasionally, the syntax-derived behaviour needs to be overridden for individual characters. The `auto-completion-override-syntax-alist` associates characters with completion behaviour and takes precedence over `auto-completion-syntax-alist`.

When customizing `auto-completion-syntax-alist`, some useful predefined settings are available. The '`Acceptance behaviour`' can be set to '`type normally`' or '`punctuation accepts`'. Surprisingly, the '`type normally`' allows you to type normally: word-constituent characters are added to the current prefix and recompleted, anything else rejects any provisional completion and inserts the character. The '`punctuation accepts`' setting causes whitespace and punctuation characters to instead accept the current completion and insert the character.

The '`Completion behaviour`' can be set to '`word`' or '`string`'. The differences between these settings is subtle, and mainly affect what happens when you start typing when the point is in the middle or end of an existing word. Essentially, the '`word`' setting will try to intelligently decide whether you wanted to add characters to the existing word or start a new word depending on where the point is, whereas the '`string`' setting will always behave as though you're starting a new word, wherever the point is.

For full control overlay syntax-related behaviour, you can manually customize the entries in `auto-completion-syntax-alist`. Each association in `auto-completion-syntax-alist` and `auto-completion-override-syntax-alist` is a two-element list of the form[1]:

```
(<accept|reject|add> <string|word|none>)
```

The first element determines what happens if there is a provisional completion at the point: `accept` accepts it, `reject` rejects it, and `add` adds the typed character to the completion's prefix. The last element determines what kind of completion is done (if any) after the typed character is inserted: `string` and `word` complete the current prefix, whereas `none` doesn't do any completion. The `string` and `word` options differ in how they decide what prefix should be completed when no completion is in progress. `string` takes the prefix to be the prefix built up by typing sequential characters. `word` takes the prefix to be the part of the current word appearing before point[2]. The difference is only evident when typing a

[1] Lisp packages can additionally set a third element that controls whether the typed character is inserted, and can replace any element in the list with a function that returns one of the values described here or nil, but these features are beyond the scope of this user manual.

[2] The value of `completion-word-thing` determines what is considered a word. Logically enough, it defaults to `word`. Lisp packages can redefine it, or even override the whole prefix-finding mechanism with their own function, but this is again beyond the scope of this user manual.

character in the middle or at the end of an existing word, in which case `string` completes the newly typed character whereas `word` completes the existing word plus the new character.

Finally, it is occasionally useful to be able to manually override a character's syntax, and have it treated one-off as though it had a different syntax class. A key binding to do this can be created using the `completion-define-word-constituent-binding` function in your '`.emacs`' file (after the line loading the predictive package, see Chapter 2 [Obtaining and Installing], page 3). This is most commonly used to allow punctuation characters to be inserted one-off as word-constituents. The bindings *M-.*, *M--*, *M-/* and *M-S-<SPC>* are defined by default in predictive mode to do precisely this for the punctuation characters '`.`', '`-`' and '`/`'.

6.2 Major Modes

The many features of predictive mode allow you to set things up appropriately for whatever language you are typing, whether it be plain text, markup languages such as LaTeX, HTML or Texinfo, programming languages such as C or Lisp, etc. Predictive mode will work happily alongside the appropriate major-mode. However, since each language makes different demands of predictive completion, you may find yourself changing a large number of settings when switching major modes.

To facilitate using predictive completion alongside different major-modes, predictive mode can run a setup function determined by the current major-mode whenever it is switched on in a buffer. Of course, you can also use major mode hooks, but hooks are less convenient if you don't want to always switch on predictive mode in that major mode, or if you want to switch it on and off whilst within the mode.

`predictive-major-mode-alist`
: Alist associating a major-mode symbol with a function, which should take one argument. The alist is checked whenever predictive mode is enabled or disabled in a buffer using the `predictive-mode` or `turn-on-predictive-mode` commands (see Section 4.2 [Basic Completion Commands], page 9). If the buffer's major-mode matches one in the alist, the associated function is called with a positive argument if predictive mode is being enabled or a negative one if it is being disabled. This makes it easier to customize predictive mode for different major modes.

Since the setup function is determined by the current major-mode, predictive mode should be switched on *after* switching to the appropriate major-mode. If you always want to use predictive mode with a particular major-mode, the easiest way to do this is to add the `predictive-mode` command to the major-mode hook in your '`.emacs`' file, using a line something like this:

```
(add-hook 'major-mode-hook 'turn-on-predictive-mode)
```

The predictive package itself includes comprehensive support for LaTeX (`predictive-setup-latex`), Texinfo (`predictive-setup-texinfo`) and HTML (`predictive-setup-html`). More contributions are always welcome!. You must ensure that the dictionaries required in order to support these major-modes (which are also included in the package, see Chapter 2 [Obtaining and Installing], page 3) can be found in your `load-path`. The LaTeX, Texinfo, and HTML support goes far beyond simply changing a few configuration variables.

See Section 6.2.1 [LaTeX Support], page 36, Section 6.2.2 [Texinfo Support], page 39, and Section 6.2.3 [HTML Support], page 41.

6.2.1 LaTeX Support

Predictive mode comes with comprehensive support for the LaTeX type-setting language. With the default settings, LaTeX support is enabled automatically when predictive completion mode is turned on in a LaTeX buffer, via an entry in `predictive-major-mode-alist` (see Section 6.2 [Major Modes], page 35).

6.2.1.1 Parsing LaTeX Documents

Predictive mode's LaTeX support parses the LaTeX file as you type (without any noticeable slow-down!), in order to identify different contexts. The main use for this information is to switch to the appropriate dictionary in different regions of a LaTeX document.

The following customization options affect this parsing:

`predictive-latex-docclass-alist`
: Alist associating LaTeX document classes (the `docclass` appearing inside `\documentclass{<docclass>}` with dictionaries.

`predictive-latex-electric-environments`
: When non-nil, environment names appearing inside `\begin{<environment>}` and `\end{<environment>}` are automatically synchronised.

By default, predictive mode will use the usual main dictionary in LaTeX mode, determined by `predictive-main-dict` (see Section 5.3 [Basic Dictionary Usage], page 24). However, by customizing `predictive-latex-docclass-alist`, the main dictionary can be selected automatically based on the document class.

In addition to the main dictionary, a number of LaTeX dictionaries are also used, grouped into four main categories: text-mode LaTeX commands, maths-mode LaTeX commands, preamble LaTeX commands, and LaTeX environments. The dictionaries in the different categories are used to look for completions in different contexts in the LaTeX document. The main LaTeX dictionaries in these categories are, respectively, `dict-latex`, `dict-latex-math`, `dict-latex-preamble` and `dict-latex-env`. In addition, there are dictionaries for LaTeX document classes (`dict-latex-docclass`), bibliography styles (`dict-latex-bibstyle`). Finally, a dictionary of cross-reference labels and dictionaries of locally defined LaTeX commands and environments, unique to each LaTeX file, are generated automatically (see Section 6.2.1.5 [Automatically Created Files for LaTeX], page 39).

Predictive mode will automatically complete words from the correct dictionary in different regions of your LaTeX document[3]. In the main body of the document it will complete from the main dictionary, as usual, and also from the dictionaries of text-mode LaTeX commands. Inside '`equation`' or other display-mode environments, between '`$`'s, or between '`\[`' and '`\]`', it will use the dictionaries of maths commands. Inside '`\begin{...}`' it will use the dictionaries of LaTeX environments. Inside '`\ref{...}`' it will use the dictionary of cross-reference labels, which is created and updated automatically for each LaTeX document. Inside '`\documentclass{...}`' and '`\bibliographystyle{...}`', it will use the document class and bibliography style dictionaries, respectively.

[3] The automatic dictionary switching is implemented using the *auto-overlays* Elisp package.

When `predictive-latex-electric-environments` is enabled, the environment name appearing inside a LaTeX `\end{<environment>}` command is automatically synchronised with its matching `\begin{<environment>}` command. The synchronisation doesn't just occur when the `\end` command is first typed; it is kept synchronised at all times, even when the `\begin` command that it originally matched is deleted, causing it to match a different `\begin` somewhere else in the document. Also, when the environment name within either a `\begin` *or* an `\end` command is modified, the environment name within its matching partner is also modified accordingly. WARNING: this feature is known to have bugs, and should probably not be used at the moment (see Section 7.1 [Known Bugs], page 43).

Predictive LaTeX mode honours the `TeX-master` variable. If it is turned on in a buffer whose `TeX-master` variable is set to the name of another LaTeX file, the `TeX-master` file will be visited, predictive mode will be enabled in its buffer, and all buffers with the same `TeX-master` will share various predictive mode settings.

The behaviour of different character syntax classes, and the behaviour of certain individual characters, is set up appropriately for LaTeX (see Section 6.1 [Character Syntax and Key Bindings], page 32). Also, a special LaTeX completion browser menu, more appropriate for browsing LaTeX commands, is used instead of the default one.

6.2.1.2 LaTeX Navigation Commands

`predictive-latex-jump-to-definition`
: Jump to the definition of whatever is at the point. If point in already on a definition, jump to the next duplicate definition of the same thing. This works for cross-references, labels, and any commands or environments that are defined in your documents preamble.

`predictive-latex-jump-to-label-definition`
: Jump to the definition of a label in the current LaTeX document. The label is read from the mini-buffer. If the point is already on the label definition, jump to the next duplicate definition of the label.

`predictive-latex-jump-to-command-definition`
: Jump to the definition of a LaTeX command in the current document. The command is read from the mini-buffer. If the point is already on the command definition, jump to the next duplicate definition of the command.

`predictive-latex-jump-to-environment-definition`
: Jump to the definition of a LaTeX environment in the current document. The environment is read from the mini-buffer. If the point is already on the environment definition, jump to the next duplicate definition of the environment.

`predictive-latex-jump-to-section`
: Jump to a section (or subsection, subsubsection, etc.) in the current document. The section name is read from the mini-buffer. If the point is already on the sectioning command, jump to the next section with the same name (if there is one).

`predictive-latex-jump-to-matching-delimiter`
: Jump to the delimiter matching the one at the point, if any. This will jump from '`\begin{...}`' to '`\end{...}`', '`\[`' to '`\]`', '`$`' to matching '`$`', and vice versa.

`predictive-latex-jump-to-start-delimiter`
: Jump to the start delimiter of the innermost environment or equation. This will jump to '`\begin{...}`', '`\[`' or '`$`'.

`predictive-latex-jump-to-end-delimiter`
: Jump to the end delimiter of the innermost environment or equation. This will jump to '`\end{...}`', '`\]`' or '`$`'.

6.2.1.3 Help with LaTeX Command Syntax

Predictive mode can automatically display a reminder of the syntax of the LaTeX command you are currently typing in the echo area. This is enabled when `predictive-latex-display-help` is non-nil (the default), disabled otherwise[4].

6.2.1.4 LaTeX Packages

Many LaTeX commands and features only become available when the appropriate package is included in the document using the '`\usepackage`' command. Predictive LaTeX mode supports this by automatically trying to load package dictionaries and configuration functions when a '`\usepackage`' command is typed, and unloading them again if it is modified or deleted.

When a '`\usepackage{`*package*`}`' command is typed, predictive LaTeX mode looks for four dictionaries based on the *package* name: `dict-latex-`*package*, `dict-latex-math-`*package*, `dict-latex-preamble`*package*, and `dict-latex-env-`*package*. These correspond to the four categories of LaTeX dictionary (see Section 6.2.1 [LaTeX Support], page 36). If any of these dictionaries are found, they are added to the list of dictionaries for the corresponding category, and will be active in the appropriate regions of the document. If the text of the '`\usepackage{`*package*`}`' command is modified or deleted, the dictionaries are removed again.

In addition, when a '`\usepackage`' command is typed, modified or deleted, predictive LaTeX mode will try to load an Elisp file called '`predictive-latex-`'*package*'`.el[c]`', and run package-specific load or unload functions, as appropriate. The variable `predictive-latex-usepackage-functions` is used to determine which function (if any) to call when loading or unloading a LaTeX package (see below). This allows arbitrary configuration changes to be made when packages are included in or removed from the document. (A common use of this is to add or remove auto-overlay regexps, see Section "LaTeX Automatic Overlays" in *Predictive Programmer Manual*.)

`predictive-latex-usepackage-functions`
: Alist associating LaTeX *package* names (strings) with a list containing two functions: a function to be called when loading the package with that name (i.e. when '`\usepackage{`*package*`}`' is typed), and a function to be called when unloading the package (i.e. when the '`\usepackage{`*package*`}`' is modified or deleted). Entries should be added to this variable by the corresponding Elisp file, '`predictive-latex-`'*package*'`.el[c]`'.

[4] The only way to add new help text definitions is to dump the relevant LaTeX dictionary to file (see Section 5.2 [Loading and Saving Dictionaries], page 22), edit it manually, and recreate the dictionary (see Section 5.1 [Creating Dictionaries], page 20).

The predictive completion package already comes with support for some LaTeX packages, though by no means all or even the most important[5].

6.2.1.5 Automatically Created Files

The predictive completion LaTeX support automatically creates and updates dictionaries of cross-reference labels, dictionaries of locally-defined text-mode and math-mode LaTeX commands (defined using '`\newcommand`'), a dictionary of locally-defined environments (defined using '`\newenvironment`' and '`\newtheorem`') for each LaTeX buffer, and a dictionary of LaTeX section titles.

All these dictionaries are saved to the directory specified by `predictive-auxiliary-file-location` (see Section 4.11 [Miscellaneous Options], page 16), so by default they will be saved to a '`.predictive/`' subdirectory of the directory containing the LaTeX file. They are given a filename of the form '`dict-latex-`*`type`*`-`*`filename`*`.elc`', where *type* is the type of dictionary ('`label`', '`local-latex`', '`local-math`' or '`local-env`'), and *filename* is the name of the LaTeX file. Note that these dictionaries are shared across all buffers with the same `TeX-master`, and the location they are saved to will be based on the `TeX-master` file's name and directory.

However, the section title dictionary is *not* saved by default, because doing so has a tendency to fail badly through hitting an internal Emacs limit hard-coded into '`print.c`'. Since the section dictionary is only used for navigation, there is little disadvantage in recreating it each time the file is loaded. However, if you do want it to be saved along with the other automatically generated dictionaries, you should first increase this internal limit by applying the '`print.c.diff`' patch (included in the predictive package) to the file '`print.c`' in the Emacs source, and recompile Emacs from the patched source. It is then safe(r) to enable `predictive-latex-save-section-dict`.

`predictive-latex-save-section-dict`
: When enabled, save the section dictionary along with the other automatically generated LaTeX dictionaries. Disabled by default because saving the section dictionary has a tendency to fail badly through hitting an internal, hard-coded Emacs limit. *Do not enable this without first patching Emacs as described above.*

To speed up loading of predictive mode's LaTeX support, a file containing information about the location of different regions within the document is saved to '`auto-overlays-`*`filename`*' also located in `predictive-auxiliary-file-location`. (Separate files are created even for buffers that share the same `TeX-master`.)

It is safe to delete any of these files, or even the entire `predictive-auxiliary-file-location` directory, as long as the corresponding LaTeX file is not loaded in Emacs at the time. They will be recreated automatically next time the file is loaded. However, if you delete the dictionary files, you will lose all learned word weights (see Section 5.5 [Dictionary Learning], page 25).

6.2.2 Texinfo Support

The predictive mode Texinfo support shares much in common with the LaTeX support, but Texinfo is a somewhat simpler markup language than LaTeX (TeXnically they're both

[5] Dictionaries and Elisp code welcome!

TEX, of course!). With the default settings, Texinfo support is enabled automatically when predictive completion mode is turned on in a Texinfo buffer, via an entry in `predictive-major-mode-alist` (see Section 6.2 [Major Modes], page 35).

6.2.2.1 Parsing Texinfo Documents

Predictive mode parses Texinfo files as you type (without any noticeable slow-down!), in order to identify different contexts. The main use for this information is to switch to the appropriate dictionary in different regions of a Texinfo document.

Four Texinfo dictionaries are provided: `dict-texinfo` containing the main Texinfo @-commands, `dict-texinfo-env` containing the environment names (things that are ended by '`@end`'), `dict-texinfo-indicating` containing just the Texinfo "indicating" commands (see Section "Indicating" in *GNU Texinfo Manual*), and `dict-texinfo-math` containing Texinfo @-commands that are specific to math-mode ('`@\`' is the only one!). In addition, `dict-latex-math` is used in math-mode[6] (see see Section 6.2.1 [LaTeX Support], page 36). Finally, a dictionary of node names, a dictionary of locally defined Texinfo commands, add a dictionary of Texinfo flags, unique to each Texinfo file, are generated automatically (see Section 6.2.2.3 [Automatically Created Files for Texinfo], page 41).

Predictive mode will automatically complete words from the correct dictionary in different regions of your Texinfo document[7]. In the main body of the document it will complete from the main dictionary, as usual, and also from the dictionaries of Texinfo @-commands. After '`@end`', it will offer environment names as completions, whereas after '`@table`', '`@vtable`', and '`@ftable`' it will offer Texinfo "indicating" commands. Inside '`@xref{...}`', '`@pxref{...}`' and '`@ref{...}`', it will use the dictionary of node names, which is created and updated automatically for each Texinfo document. Inside '`@clear{...}`', '`@ifset{...}`' and '`@ifcler{...}`', it will use the automatically-generated dictionary of flag names. Inside '`@math{...}`' it will offer standard TEX maths commands as completions[8].

The behaviour of different character syntax classes, and the behaviour of certain individual characters, is set up appropriately for Texinfo (see Section 6.1 [Character Syntax and Key Bindings], page 32). Also, a special Texinfo completion browser menu, more appropriate for browsing Texinfo commands, is used instead of the default one.

6.2.2.2 Texinfo Navigation Commands

`predictive-texinfo-jump-to-definition`

> Jump to the definition of whatever is at the point. If point in already on a definition, jump to the next duplicate definition of the same thing. This works for cross-references, node names, flags, and any commands that are defined in your documents preamble.

---

[6] It ought to be `dict-tex-math`, but that will have to await TEX support. See Section 7.1 [Known Bugs], page 43.

[7] The automatic dictionary switching is implemented using the *auto-overlays* Elisp package.

[8] Currently, it actually offers LATEX maths commands, see Section 7.1 [Known Bugs], page 43.

`predictive-texinfo-jump-to-node-definition`
: Jump to the definition of a node in the current Texinfo document. The node is read from the mini-buffer. If the point is already on the node definition, jump to the next duplicate definition of the node.

`predictive-texinfo-jump-to-command-definition`
: Jump to the definition of a Texinfo command in the current document. The command is read from the mini-buffer. If the point is already on the command definition, jump to the next duplicate definition of the command.

`predictive-texinfo-jump-to-flag-definition`
: Jump to the definition of a Texinfo flag in the current document. The flag name is read from the mini-buffer. If the point is already on the flag definition, jump to the next duplicate definition of the flag.

6.2.2.3 Automatically Created Files

The predictive completion Texinfo support automatically creates and updates dictionaries of node names, dictionaries of locally-defined Texinfo commands (defined using '`@macro`', '`@rmacro`' or '`@alias`'), and a dictionary of flags (defined using '`@set`'), for each buffer.

All these dictionaries are saved to the directory specified by `predictive-auxiliary-file-location` (see Section 4.11 [Miscellaneous Options], page 16), so by default they will be saved to a '`.predictive/`' subdirectory of the directory containing the Texinfo file. They are given a filename of the form '`dict-texinfo-`*`type`*`-`*`filename`*`.elc`', where *type* is the type of dictionary ('`node`', '`local-texinfo`', or '`flag`'), and *filename* is the name of the Texinfo file.

To speed up loading of predictive mode's Texinfo support, a file containing information about the location of different regions within the document is saved to '`auto-overlays-`*`filename`*' also located in `predictive-auxiliary-file-location`.

It is safe to delete any of these files, or even the entire `predictive-auxiliary-file-location` directory, as long as the corresponding Texinfo file is not loaded in Emacs at the time. They will be recreated automatically next time the file is loaded. However, if you delete the dictionary files, you will lose all learned word weights (see Section 5.5 [Dictionary Learning], page 25).

6.2.3 HTML Support

HTML is a simpler markup language than LaTeX or Texinfo, and the predictive mode support for it is correspondingly simpler. It includes a dictionary of HTML tags, `dict-html`, and dictionaries of valid attributes for different tags, `dict-html-`*`tag`*. Completion is context-sensitive: typing a '`<`' character will start completing an HTML tag, whereas within a tag predictive mode will complete from the appropriate tag attribute dictionaries. With the default settings, HTML support is enabled automatically when predictive completion mode is turned on in an HTML buffer, via an entry in `predictive-major-mode-alist` (see Section 6.2 [Major Modes], page 35).

To speed up loading of predictive mode's HTML support, a file containing information about the location of different regions within the document is saved to '`auto-overlays-`*`filename`*' located in `predictive-auxiliary-file-location` (see

Section 4.11 [Miscellaneous Options], page 16), which by default is a '`.predictive/`' subdirectory of the directory containing the HTML file.

It is safe to delete this file, or even the entire `predictive-auxiliary-file-location` directory, as long as the corresponding HTML file is not loaded in Emacs at the time. They will be recreated automatically next time the file is loaded. However, if you delete the dictionary files, you will lose all learned word weights (see Section 5.5 [Dictionary Learning], page 25).

7 Bugs and Future Improvements

The predictive completion package has been tested on Emacs versions 21.3, 22.x and the current CVS version, and is known to work reasonably well. It will not work under older versions, and currently doesn't work under any version of XEmacs.

Report bugs and feature requests to `toby-predictive@dr-qubit.org`. Even reports of which versions of Emacs it runs under are useful at this stage (though check the website `http://www.dr-qubit.org/emacs.php` first to make sure your version is not already listed).

7.1 Known Bugs

Known bugs (in no particular order):

1. Predictive mode does not work under XEmacs. Making it work will take some compatibility work, but should be possible.
2. The `predictive-latex-electric-environments` feature is very flaky, and should probably not be used at the moment. It has occasionally been known to completely mess up LaTeX files by randomly deleting parts of them!
3. There are probably still bugs remaining in the `auto-overlay` code, used in some major modes to set up region-local dictionaries. If you find one, please report it, along with precise instructions on how to reproduce it. They're fiendishly difficult to track down!
4. The predictive mode Texinfo support uses `dict-latex-math` inside '`@math{...}`', when it should only use TeX maths commands.

7.2 Future Improvements

Possible future improvements to predictive completion (in no particular order):

1. Allow the learning rate (i.e. increment added each time a word is learnt) to be set buffer-locally and/or separately for each dictionary?
2. Support for more major modes. (This one will probably be here for ever. I suspect new major modes are being created faster than predictive mode setup functions!)
3. More predictive mode configuration variables could be allowed to have "overlay-local" bindings, not just the active dictionary, `completion-menu` function, and `completion-word-thing`.
4. Predictive mode should be integrated with the *semantic* or similar package, to make it more useful for programming languages. At the moment, it works best with plain text and markup languages.

8 Credits

Much inspiration for the Emacs predictive completion package came from a similar package written for the *nedit* editor by Christian Merkwirth. Most significantly, it provided the clue that ternary search trees are the best data structure to use for the dictionaries.

Ternary search trees are described in a very readable article by Jon Bentley and Robert Sedgewick. The article can be found via http://www.cs.princeton.edu/~rs/strings/.

Finally, the English dictionary supplied with the predictive completion package was originally based on the British National Corpus frequency tables, available from http://www.itri.brighton.ac.uk/~Adam.Kilgarriff/bnc-readme.html.

Appendix A Command Index

Appendix B Variable Index

A

C

P

Appendix C Concept Index

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W

Appendix D Copying this Manual

D.1 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

D.1.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.